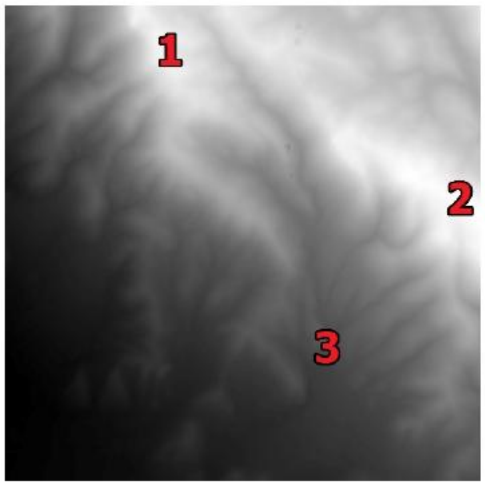
Assignment 4

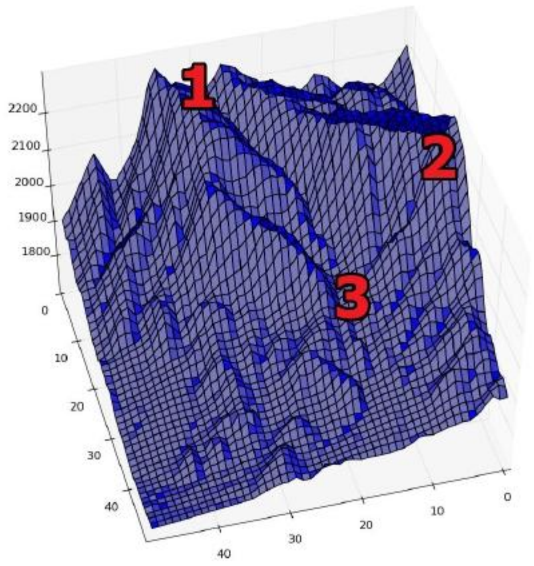
In this assignment you are going to write functions to visualize the surface runoff across the mountains in the area of Heber City, Utah. Your solution to the problems in this assignment will be in two files. A4Func.py will contain the definitions for all of the functions described below. A4.py will call those functions to produce specific results. The result of this assignment will be a number of images and print statements.

You're going to be using loops, lists and tuples to analyze an array of data and create interesting visualizations.

Watershed

The watershed (or drainage basin) is an area of land where water from rain or melting snow drains downhill into a body of water. The watershed includes both the terrain over which the water flows as well as the actual rivers and streams which transport the water downwards. The watershed acts like a funnel; water seeks the fastest route downhill. We can model this phenomenon using a terrain elevation map.





A terrain elevation map (sometimes called a height map) is a representation of actual terrain. It is a 2D array where every element is an elevation measurement from actual terrain. The samples are uniformly distributed along latitudinal and longitudinal lines. We can visualize a height map in several ways. We can visualize it as a 2D image, where the highest point is colored white and the lowest is colored black. Alternatively, we can make a 3D plot where the height of each value is shown in a 3D image. The two images below illustrate this with corresponding points labeled. In this assignment, we'll be using the 2D, grey-scale visualization approach.

Watershed functions

In script A4Func.py you are going to have to provide function definitions for the following functions. The script A4Func.py already includes place holders for these functions. Currently, each function returns **`None`**. It is your task to write the expressions to accomplish the indicated functionality.

1. **`findLowNhbr( terrain )`**
   Inputs: **`terrain`** which is a 2D array representing a terrain elevation map.
   Outputs: Two 2D arrays. One represents the offset in rows toward the direction of greatest downward change. The other represents the offset in columns toward the direction of greatest downward change.
   Functionality: Water flows in the direction where the terrain is the steepest. In our terrain model, we need to determine, at each point, which direction the terrain drops the most.

Because our terrain model is a uniformly sampled 2D array of elevation measurements, every point, P, in our data has eight points surrounding it (except for those on the edge. We'll treat them specially.) So, our directions are limited to nine directions: the direction to each of the neighbors as well as the point itself since it is possible for a terrain value to be in the bottom of a bowl and lower than all of its neighbors. We can describe that direction as the offset in the row and the column to the cell that has the lowest elevation. So, if the lowest point in the neighborhood of P is down and to the right of P, the offset would be +1 row and +1 column. Similarly, if the lowest point were to the left, the offset would be +0 row and -1 column. And if P is the lowest point in the neighborhood, then the offsets would be 0 and 0 for both row and column. We want the output arrays to contain these offsets.

(-1,-1)	(-1,0)	(-1,1)
(0,-1)	P	(0,1)
(1,-1)	(1,0)	(1,1)

Imagine that we call the output arrays **`rowOffset`** and **`colOffset`**, representing the row offset and column offset values, respectively. Then for a point in the terrain at index **`[i,j]`** the index in the terrain of the neighboring cell lowest in its neighborhood is **`[i+rowOffset[i,j], j+colOffset[i, j]]`**. We will be able to use these arrays to find which way the water flows.

Because the boundary points don't have the full set of neighbors, we will simplify the problem by saying that on the boundaries, the row and column offsets should be zero. In other words, we're assuming that the boundary points are all their own lowest neighbors. So, the results of this function are two 2D arrays the same shape as the terrain array. The values on the edges of the arrays are 0, and every other entry should be a -1, 0 or 1.

Doing this function is going to require nested loops. First use one loop to iterate through each row. For each row, use another loop to iterate through the columns. This is the point whose neighborhood you're testing. You might need other loops to actually test the neighborhood.

2. **`findPits( terrain )`**
   Inputs: **`terrain`** which is a 2D array representing a terrain elevation map.
   Outputs: An N-by-2 array of integer pairs representing the indices (row, col) of all of the "pits" in the terrain.
   Functionality: A "pit" in the terrain is, mathematically speaking, a local minimum. It is the point in the terrain that is lower than all of its neighbors. This function finds all of the points in the terrain map (ignoring the boundary points) and returns the row, column index values of each pit.

We can use the two arrays produced by **`findLowNhbr`** to compute pits. A pit is everywhere in the terrain except the boundaries where the row and column offsets to the lowest neighbor is 0. This means, of course, that this point is the lowest value in its neighborhood. So you should find out indices where both row and column offsets are zero.

To implement this function, you might find the function **`np.where`** useful. This function returns an array of indices which correspond to the non-zero values in an array. However, it returns a tuple of arrays, one per dimension.

In one-dimension case,

```
>>> offsetX = np.array([0, -1, 1, 0, 0, 1])
>>> np.where(offsetX == 0)
(array([0, 3, 4] ),)     # a tuple with one array in it
```

In two dimensions, it returns a tuple with two arrays. The i-th element in each array contains the row and column indices for an entry in the array which satisfied the "where" condition.

```
>>> offsetX = np.array([[0, -1, 1, 0, 0, 1],
                        [1, -1, 0, 1, 0, 1],
                        [0, 0, 1, 0, -1, -1]])
>>> indices = np.where(offset == 0)
>>> indices
(array([0, 0, 0, 1, 1, 2, 2, 2]), array([0, 3, 4, 2, 4, 0, 1, 3]))
# a tuple with two arrays
```

Here are the positions of my pits. You should make sure you have the same pits.

[5, 207], [11, 213], [13, 185], [13, 226], [14, 217], [23, 183], [32, 288], [88, 176], [88, 178], [101, 180], [107, 180], [127, 2], [134, 1], [138, 2], [153, 4], [167, 16], [174, 23], [182, 21], [198, 32], [225, 42], [242, 298], [245, 6], [246, 1], [250, 29], [251, 55], [252, 10], [258, 173],

[259, 37], [261, 166], [262, 92], [263, 6], [264, 156], [266, 11], [266, 14], [267, 7], [268, 84], [268, 87], [269, 10], [269, 13], [271, 2], [271, 5], [273, 150], [274, 1], [274, 11], [275, 73], [276, 130], [276, 214], [277, 40], [281, 65], [282, 2], [283, 40], [285, 55], [286, 46], [286, 50], [286, 68], [289, 54], [297, 84]

3. **`traceDrop( terrain, row, column )`**
   Inputs: **`terrain`** which is a 2D-array representing a terrain elevation map.
   `row` indicating the row index of the starting point of the drop.
   `column` representing the column index of the starting point of the drop.
   Outputs: An N-by-2 array of integer pairs representing the path a raindrop would follow down the terrain from the initial point (row, column). The columns are the row and column indices of all the points in the terrain that make up the path.
   Functionality: This function traces the path of a drop of water down the terrain. As with **`findPits`**, we're going to use the function **`findLowNhbr`**.

We can construct a path by using a list. The first point in our path is the position given as the start position. We use the row and column offset arrays from **`findLowNhbr`** to figure out which neighboring cell we should move to from this cell. Every time we use the offsets to compute the next element index, we add it to the path. We continue doing this until we end up in a pit (where the row and column offsets are zero).

This function uses a while loop. The nature of this loop is indefinite. You do not know how long the path is; it depends on where you start. However, you know the loop condition. You want to keep following the terrain downwards until you reach a point that's lower than all of its neighbors.

To construct the path, you'll use the list data type. The list data type allows you to add new values to the list so you can increase the contents dynamically, something you can't do with Numpy arrays. You can create an empty list and add pairs of row, column values by using the append method. Finally, you can convert the list to a Numpy array.

Exercising functions

In the file A4.py, we're going to exercise the functions you've created by calling them with an input and saving the results. The template script A4.py already imports Numpy, Pylab and A4Func. In addition, it reads in the terrain file "elevation.npy" and saves the 2D array in a variable called **`terrain`**. This is a large array and it might be easier to test your code on a smaller array. I've also added a smaller 10-by-10 simulated terrain called "test.npy". Changing the line that creates the terrain variable to load the other file will give you access to the simpler file -- but all of your code will use the same variable. So, changing it back to the full terrain is as simple as changing which file is loaded.

1. Draw the terrain into a figure with the command

```
pylab.imshow( terrain, cmap = pylab.cm.gray )
```

which draws the terrain array as an image. The **`cmap`** argument tells Python to use a gray colormap to color the data. The smallest values will be black and the largest values will be white.

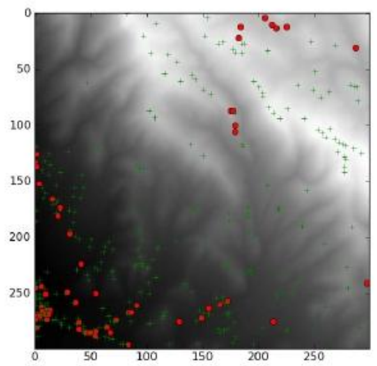
2. Compute the pits of the terrain by calling **`findPits`** on **`terrain`** and draw locations of the pits on top of the image as red dots (format string **`'r.'`**). Remember that **`findPits`** returns an N-by-2 array of row and column indices. But when you plot them, the rows are the y values and the columns are the x value. Make sure you use the right x and y values when you plot the red dots.

3. Compute the peaks of the terrain. You can use your **`findPits`** function to find the pits by calling the function on the negation of **`terrain`**. Plot peak locations on top of the pits as green crosses (Python format string **`'g+'`**).

4. Print the number of peaks and the number of pits. In your code, if you simply type in a literal number, even if the number is correct, you will lose points. Your program should compute the values and have them stored in a variable that you output. I've listed here my actual answers so you can see if you're getting the right values.

```
Number of peaks: 273
Number of pits: 57
```

5. Save this figure as "peaksNPits.png" and then call **`pylab.close()`** so that this figure does not interfere with the next stage. Below shows my saved figure.



6. Use **`pylab.ginput(1)`** to let the user click on a single point in the terrain. We will use this point to test **`traceDrop`**. The point the user clicks is the starting point you'll pass to the **`traceDrop`** function. The function call **`pylab.ginput(1)`** will give you a tuple consisting of one tuple with two values (the x and y values). The x and y values won't necessarily be integers but you need integers to index into the array. The following code will

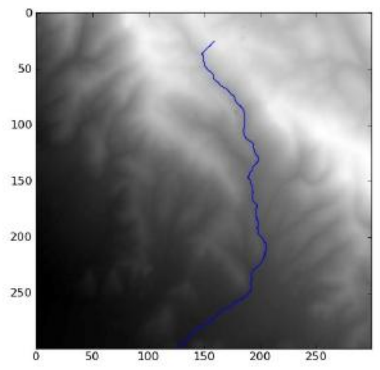
turn the result of `pylab.ginput(1)` into a row and column index.

```
pylab.imshow( terrain, cmap = pylab.cm.gray )
point = np.array( pylab.ginput(1)[0] )
r = int( np.round( point[1] ) ) # an integer row index
c = int( np.round( point[0] ) ) # an integer column index
```

7. Draw the terrain into a figure like in step 1.

8. Call **`traceDrop`** with the row and column index from step 6. Plot the path **`traceDrop`** returns as a blue line on top of the image. Watch out for the row-column/x-y issue discussed in step 2.

9. Save this figure as “raindrop.png” and then call **`pylab.close()`**. The image should be similar to the image below. It won't be exact, because it depends on where you click. In this case, I clicked on the white pixel at the top end of the path.



Grading notes

If your script doesn't run, you automatically lose 50% of the points. Make sure that your script runs without any errors.